"Plug & Test" at System Level via Testable TLM Primitives

Homa Alemzadeh[1], Stefano Di Carlo[2], Fatemeh Refan[1], Paolo Prinetto[2], Zainalabedin Navabi[1]

[1]CAD Research Group
ECE Department
University of Tehran, Tehran 14399, Iran
{*homa, refan, navabi*}*@cad.ece.ut.ac.ir*

[2]Control and Computer Engineering Department
Politecnico di Torino, I-10129 Torino, Italy
{*stefano.dicarlo, paolo.prinetto*}*@polito.it*

Abstract

With the evolution of Electronic System Level (ESL) design methodologies, we are experiencing an extensive use of Transaction-Level Modeling (TLM). TLM is a high-level approach to modeling digital systems where details of the communication among modules are separated from the those of the implementation of functional units. This paper represents a first step toward the automatic insertion of testing capabilities at the transaction level by definition of testable TLM primitives. The use of testable TLM primitives should help designers to easily get testable transaction level descriptions implementing what we call a "Plug & Test" design methodology. The proposed approach is intended to work both with hardware and software implementations. In particular, in this paper we will focus on the design of a testable FIFO communication channel to show how designers are given the freedom of trading-off complexity, testability levels, and cost.

Index Terms *— Design for Testability (DFT), System Test, Transaction Level Modeling (TLM), System Level Design*

1. Introduction

With the increasing complexity of digital systems, and the reduced time to market, Electronic System Level (ESL) design has rapidly emerged as the main design methodology for implementing large digital systems, with a wider and wider usage of Transaction Level Modeling (TLM).

TLM is a transaction-based approach to modeling digital systems. It separates the details of communication among modules from the implementation of functional units. In the TLM notion, communication mechanisms such as busses or FIFOs are modeled as abstract channels accessed through interface functions. Transaction requests between modules take place by calling these functions, which encapsulate low-level details of the information exchange. At the transaction level, the emphasis is more on data transfer functionalities rather than on their actual implementations [1].

Although in the recent years digital system design has moved from gate and register transfer level to higher abstraction levels such as TLM, testing and testability techniques are still mostly performed at low abstraction levels, and mainly at the gate level. System level designers need new tools to insert test and testability requirements at the TLM level in a completely transparent and automatic way, without concerning themselves with the details and intricacies of lower levels. These tools will play at system level the same role that Electronic Design Automation (EDA) tools play at gate and RT levels.

Test and testability requirements at TLM include those for computational modules as well as the communication channels. The added testing capabilities have to be designed in order to be automatically synthesized, later in the design process, either into hardware or software modules according to the designer's choices and needs.

This paper presents the preliminary results toward the definition of a design methodology capable of guaranteeing the implementation of "Plug & Test" modules and communication channels. Different high-level testing strategies and functional procedures are proposed to verify the proper functionalities of TLM channels with the goal of building a library of "Testable TLM Primitives" characterized by having a specific degree of testability and Built-in Functional Self Test (BIFST). In particular, in this paper we will focus on the *tlm_fifo* as a basic communication channel in the SystemC TLM primitive channels library. The added testing capabilities provide designers with the freedom of trading-off complexity, testability, and cost in the early stages of the design. To evaluate the quality of the proposed test strategies, high-level coverage metrics are introduced based on a behavioral model of the communication channel.

The paper is organized as follows: Section 2 briefly presents an overview of TLM whereas Section 3 overviews related works. Section 4 describes the proposed testing approach at the TLM level. Section 5 introduces the *tlm_fifo* channel and presents a model for its behavior. The testable *tlm_fifo* and the procedure for testing its different functions are proposed in Section 6. In Section 7 we evaluate the quality of the proposed testing capabilities in terms of complexity, testability, and cost. Finally Section 8 concludes the paper.

2. Transaction Level Modeling

This section gives a brief overview of the evolution of abstraction levels in digital system design over the past

1-4244-4203-0/08/$20.00

fifty years. The concepts introduced in this section represent the motivation for the use of Electronic System Level (ESL) tools in design of next generation hardware.

Every fifteen to eighteen years, digital design techniques move to a higher abstraction level (Figure 1). With this change, new tools and methodologies evolve, and intermediate steps taken when moving to a higher abstraction level become sample points that help us decide on future trends.

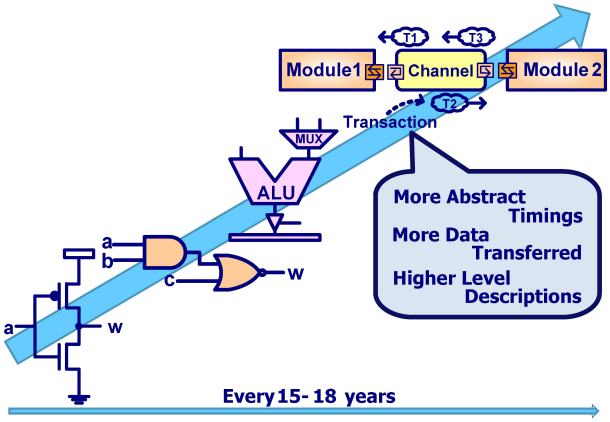

Figure 1 Design Evolution - Every 15 to 18 Years

Transistor level design of logic circuits that represented the main design techniques fifty years ago involves wiring individual transistors to form gates and Boolean functions. At this level, interconnections are simple wires that serve as carriers between various transistors. Data flowing in these carriers are described at a very low abstraction level with very little functionalities associated with this description. By moving to the next abstraction level represented by the gate level we start to have components (gates) for which a clear functionality can be easily defined. Interconnections at this level are Boolean signals that carry binary data between gate level components.

Moving to higher abstraction levels, the RT level models interconnections as word length busses, and components as RT level functional blocks such as ALUs and register files. Interconnections at this level carry far more information and have structures that are far more complex than those at the gate level.

The evolution of design abstractions has now got to its next upper level that is the System Level. At this level interconnections become complex busses or switches and the components become complex processing elements that are often themselves complex RT level systems. Designing at the System Level involves defining the functionality of individual processing elements and defining their communications with abstract channels.

Transaction Level Modeling (TLM) is regarded as the next step in the direction of system level design. TLM is a transaction-based modeling approach originally based on high-level programming languages such as C++ and SystemC. It emphasizes on separating communications from computations within a system. In the TLM notion, computation units are modeled as modules with a set of concurrent processes that calculate and represent their behavior. These modules communicate in the form of transactions through abstract channels [2]. Based on the model accuracy and the incorporation of timing details into the design, we can identify three classes of TLM: Un-timed, Approximately-timed and Cycle Accurate. Un-timed TLM level ignores many details and timing annotations in design specification, and considerably decreases the number of events to be processed by a simulator. This provides a faster simulation than lower modeling levels such as RT and gate levels.

3. State-of-the-art

Design for Testability is a set of design techniques employed to ensure device testability. DFT is generally used to reduce testing costs (including generation and application), and to enhance the test quality (fault coverage) [3]. Different from the design techniques evolution that moved to the TLM abstraction level, most of the testability techniques used in the recent years are still inserted at the gate or register transfer level.

One of the most popular gate level DFT techniques at the gate level is the scan design. Among different scan-based designs, partial-scan gained the major attention in both industry and research area. In the partial scan approach a subset of the flip-flops of a design are used to form scan chains used to reduce the test complexity. [4] categorizes Partial scan designs into three main groups: testability analysis based [5,6], structural analysis based [7,8], and test generation based [9, 10, 11]. Another popular gate level Testability solution is test points insertion, which improves testability by adding control and observation points to the circuit [12, 13, and 14].

Although gate-level techniques are usually applicable at the RT level, there are testability studies that are specifically focus on RT level circuit test. For example, in order to improve fault coverage, [15] and [16] used some testability measures. In a number of researches, hierarchal test generation [17] based DFT method for testing Datapath and Controller at RT level [18, 19, and 20] has been used. There are also a number of works in RT level test point insertion including BIST based [21, 22, 23], and non-BIST based [24] methods. Furthermore there are researchers which apply test synthesis at RTL, including [25, 26].

To cope with the change in design complexity from wiring gates to interconnecting complex processing elements at TLM abstraction level, new DFT approaches are required. TLM designers like to avoid concerning themselves with details and implementations of lower levels. The challenge is thus developing tools for automatic insertion of testing capabilities at the TLM level.

To our best knowledge, this work is the first attempt to tackle Design for Testability and BIST issues at the TLM

level by incorporating testing facilities and automatically adding testability features into the TLM primitive channels. It is worth mentioning that the added testing capabilities will then be synthesized either in hardware or software modules, according to the designer's choices and needs.

4. TLM Testing Methodology

Testing strategies at TLM have to be defined at a very high abstraction level, even before deciding whether the communication channels will be implemented in hardware, in software, or in a proper mixing of both. The challenge here is to allow the designer to automatically include testing capabilities in the design at the very early stages of the design process. Obviously, since we are working at a very high abstraction level, typical design for testability approaches targeting structural fault models (e.g., scan chains) cannot be applied. Actually, the concept of hardware components still has to be defined at this level. Thus the only possible and reasonable solution to guarantee a certain level of testability at TLM level is resorting to pure functional testing.

Test and testability requirements include those for computational modules as well as the communication mechanisms. In this work we will focus on the testability of TLM primitive channels and on the development of approaches for testing their functionality. We consider a typical portion of a system modeled at TLM composed of two functional modules called Writer and Reader and a single communication channel connecting the two modules (Figure 2). The Writer is in charge of sending information to the Reader over the defined channel.

Figure 2 System Architecture: Writer-Channel-Reader

This architecture can fit any type of TLM communication channel; nevertheless, for the sake of simplicity in this paper we will focus on the basic TLM primitive, *tlm_fifo* which implements the unidirectional TLM communications.

The overall idea is to define for each TLM primitive channel, based on its behavioral model, a suitable functional test strategy. This strategy should allow the Communication Channel (CC) to test its behavior regardless of the actual final implementation. To achieve this goal, we introduce the concept of "*Plug and Test*" at system level design. We provide a design methodology capable of adding test functionalities to the blocks composing a TLM design to be translated later on into *Built-in Functional Self Test* (BIFST) facilities in the final product. In particular, our design methodology comprises the idea of enriching each computation unit of a design with predefined Test Facilities and replacing each original communication channel with its corresponding new BIFST-able version.

A basic test architecture at the TLM level is proposed in Figure 3. In this architecture, each computation unit as well as the communication channel is modified to include the required BIFST facilities. From a practical point of view, this requires adding some new classes and methods to the modules and communication channel.

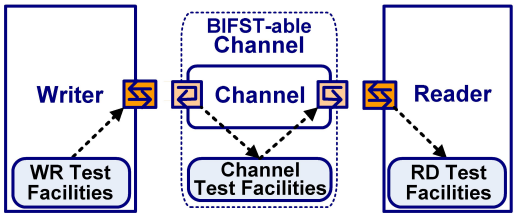

Figure 3 Test Architecture: Test Facilities Added

The added BIFST facilities can be implemented by instantiation of different BIFST units including Test Data Generators (TDG), Test Response Evaluators (TRE), and Test Controllers as depicted in Figure 4. In addition, some Interfaces are needed to support the communication of modules and controllers with the rest of system for the management of test execution.

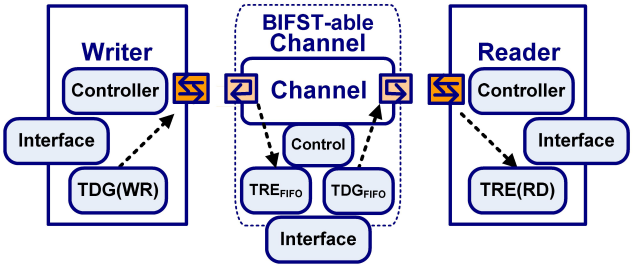

Figure 4 Test Facilities Implementation

The proposed test architecture can support three different testing strategies:

1. ***Transaction Testing***: Testing the transactions between the channel and Writer/Reader separately. This includes:
   a. ***Write Transactions***: Testing the functionality of the interconnection between the Writer and the Channel without using the Reader;
   b. ***Read Transactions***: Testing the functionality of the interconnection between the Channel and the Reader without using the Writer;
2. ***Channel Self-Testing***: Testing the Channel as an isolated component, without considering its connections with the Writer and the Reader;
3. ***Integration Testing***: Testing the integration between Writer, FIFO and Reader.

Each of these testing strategies requires the implementation of a set of different functional blocks in

the test architecture of Figure 4. For example the implementation of Write Transaction Test strategy consists of the definition of the TDG and Controller blocks of the Writer as well as the TRE and Controller units inside the channel.

To implement the above testing strategies, different working modes are also required for the computation units and the communication channel:

- Communication Channel Working Modes:
  - *Normal Mode*
  - *Write Transaction Test Mode*
  - *Read Transaction Test Mode*
  - *Self-Test Mode*
- Computation Units Working Modes:
  - *Normal Mode*
  - *Blocked Mode* (Waiting for Channel to perform Write/Read or Test)
  - *Write/Read Transaction Mode*
  - *Integration Test Mode*

In the sequel of this paper, we will focus on the Transaction Test Strategy while leaving the Self Test and Integration Test Strategies for future works.

Transaction Testing Strategy has several advantages. First of all the "write" and "read" functionalities of the channel can be easily tested autonomously (e.g., testing the writing functionalities does not require the reader to be involved). This minimizes the actors involved during the test and consequently the impact of the test on the normal behavior of the system. Also since the reader and the writer are not concurrently involved in the test, their related activities do not need any ad-hoc timing and synchronization. Each of them has the responsibility of testing the functionalities of the channel actually used. The main drawback of this approach is that the communication channel and the computation units have to be modified to include the BIFST facilities leading to some overhead.

5. *tlm_fifo* Primitive

This section introduces the *tlm_fifo* and provides a description of its functionalities that will be used in the rest of the paper as a starting point for the definition of a testable version of this primitive.

SystemC TLM standard [27] released by OSCI is currently the most widely-used approach to transaction level modeling. The reason of choosing SystemC for modeling at transaction level is the close correspondence of SystemC with lower RT level descriptions, and its high level interface with C++. SystemC is a class library based on C++, an object-oriented language extensively used by software developers. It implements main hardware-oriented parameters like Time, Concurrency, and Hardware data types [28].

The basis of SystemC TLM is on classes and methods for modeling bidirectional, unidirectional, blocking and non-blocking communications. Interface classes form the heart of the SystemC TLM standard and TLM primitive channels are the implementations of these interfaces. Core TLM interfaces presented in this standard include:

- The Unidirectional Blocking Interfaces:
  - *tlm_blocking_get_if*
  - *tlm_blocking_peek_if*
  - *tlm_blocking_put_if*
- The Unidirectional Non Blocking Interfaces
  - *tlm_nonblocking_get_if*
  - *tlm_nonblocking_peek_if*
  - *tlm_nonblocking_put_if*
- Bidirectional Blocking Interface
  - *tlm_transport_if*

The basic TLM primitive channel is *tlm_fifo*, a class which implements all the unidirectional interfaces mentioned above. Other primitive TML channels including *tlm_req_rsp_channel* and *tlm_transport_channel* implementing bidirectional interfaces are based on *tlm_fifo*.

The implementation of *tlm_fifo* is based on the implementation of SystemC *sc_fifo* with the additional capability of having a zero or infinite size [29]. The class description provides access methods for Write to, Read from, Peek, Resize and Debug the *tlm_fifo*.

The *tlm_fifo* primitive in addition to being a channel for implementing TLM unidirectional communications is also used as a primitive in more advanced TLM channels. Therefore making this channel testable is useful for direct use of *tlm_fifo*, and at the same time for providing testability for more complex channels.

The TLM specification itself is not formal enough to systematically enable the development of a complete test and to enable the evaluation of its coverage. Therefore the first step toward the realization of a testable TLM channel is its representation using a more formal model. It is necessary to identify a suitable formal representation capable of capturing a wide variety of TLM specifications. For this purpose we decided to adopt the UML state charts. UML state charts allow an effective representation of the communication channel behavior. Moreover, they can be easily inserted in a more complex UML model allowing the description of additional information and eventually automating test generation activities.

Figure 5 shows a UML state chart modeling the functionality of the *tlm_fifo* channel. Labels on the arcs (transitions in the UML formalism) correspond to the execution of the FIFO methods. Transitions can be conditioned through the use of guards reported in square brackets. For the sake of readability, transitions with multiple labels are equivalent to multiple transitions, each one fired by one of the labels displaced on the arc.

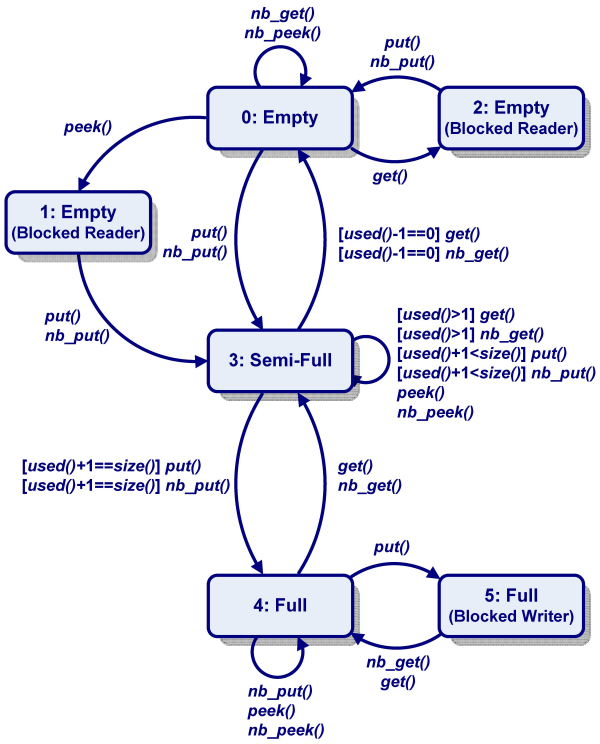

Figure 5 *tlm_fifo* UML State Chart

The *tlm_fifo* has three main working states:

- ***Empty:*** the buffer associated with the FIFO does not contain any information;
- ***Full:*** the buffer associated with the FIFO is completely full;
- ***Semi-Full:*** the FIFO is partially full, it contains information but its buffer is not completely full.

The FIFO evolves among these three states depending on the call of its *put(), get()*, and *peek()* (blocking and non-blocking (*nb_*)) methods from the modules connected to its communication ports.

In particular, the *put()* method allows writing a new element into the FIFO, the *get()* method reads an element from the FIFO (removing the element after the read) and finally the *peek()* method reads an element from the FIFO but do not remove the element from the buffer. These three methods are blocking methods, meaning that in case they cannot complete their execution the caller is blocked until the failing condition is resolved. For example trying to call a *get()* or *peek()* method on an empty FIFO will block the caller until an element will be written in the channel. This situation is modeled by the two states labeled with *Empty (blocked reader).* We have to distinguish between the call of a *get()* and the call of a *peek()* since the status where the FIFO is resumed when an element is written is different in these two cases. The same situation happens when trying to write a new element in a full FIFO. In this case the writer will be blocked and the FIFO will be pushed in state *Full (blocked writer)* waiting for an element to be read from the channel. The non-blocking versions of the *put()*, *get(),* and *peek()* methods are *nb_put()*, *nb_get(),* and *nb_peek(),* respectively. They behave as the blocking versions but do not block the caller in case of failure.

The condition of FIFO full or empty is checked through the guards placed on some of the transitions. These guards resort to the methods *used()* and *size()* provided by the *tlm_fifo*, that return the number of elements already in use and the maximum allowed elements respectively.

To conclude, The model also includes the execution of other non-blocking methods like *nb_can_put()*, *nb_can_get()*, and *nb_can_peek()* which are not shown on the diagram because they do not change the state of FIFO.

6. Test Strategy Implementation

The architecture proposed in Section 4 is general enough to implement different test strategies. In this section we show the implementation of the Transaction Test Strategy providing a sample of a test program, and the related implementations of the testable *tlm_fifo*. The test approach we adopt here is based on the FSM model of *tlm_fifo* communication channel presented in Figure 5. We implement the test strategy by going through the following steps:

- Defining a test procedure able of testing the functionalities of each Transaction (method call implemented by the *tlm_fifo);*
- Identifying the additional test functionalities the *tlm_fifo*, the Writer, and the Reader should provide to realize the testing procedure defined before;
- For each additional functionality, defining the corresponding SystemC implementation.

For the sake of readability, in this paper we will show a single example of testing procedure for the *put()* method involving the Writer and the FIFO, together with the corresponding SystemC TLM descriptions.

Before detailing the actual test procedure we need a preliminary consideration. The *put()* method allows writing a data *X* into the FIFO. Working at the TLM abstraction level the semantic of *X* is undefined (e.g., a single bit, a *jpeg* image, an NoC packet, an *mp3* file, etc.). In order to be as general as possible, our test procedures do not take into account the semantic of the data associated with *X*, considering it as an abstract element. In a real implementation the proposed testing solution should be completed with additional operations designed to test the consistency of the particular data type transmitted over the channel and to the specific type of faults affecting this data.

Figure 6 shows the test procedure for a *testable_tlm_fifo* with a four-element buffer. The test procedure is depicted using an UML sequence diagram showing the Writer and

the FIFO as actors performing the sequence of actions. The overall idea in developing this procedure is trying to stress the *put()* method in different operational conditions and verifying its correct behavior.

The Writer starts the test procedure by issuing *put()* transactions. Each of the written values has to be verified once written in the FIFO. Here we encounter the first testing requirement for the *testable_tlm_fifo*: we need to be able to read the content of the most recently written element, (without modifying the actual content of the internal buffer) and we need to compare this value with the expected one. This requirement leads to the definition of the following two test facilities:

- ***t_peek():*** performs a peek operation, i.e., reads the last written element internally to the FIFO. After the execution of the operation the content of the FIFO is unchanged;
- ***t_compare(A,B):*** compares the two values A and B and returns a Boolean result.

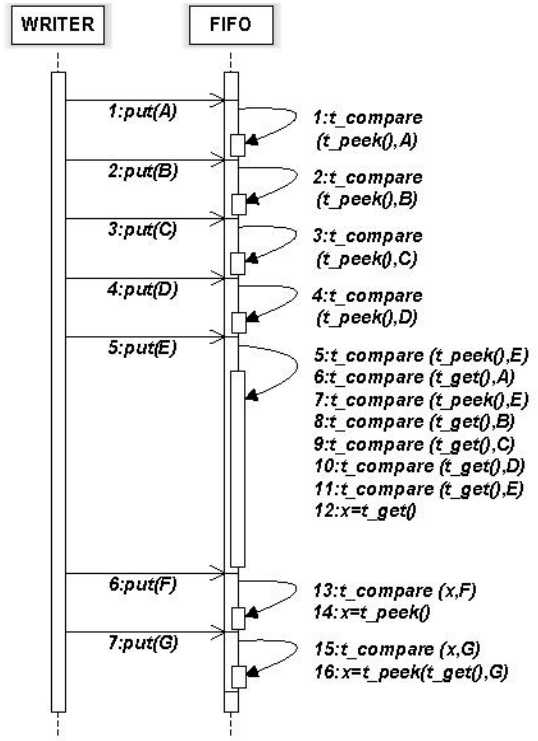

Figure 6 Test Procedure for the *put()* Method

These testing methods are used during steps 1 to 4 in the FIFO side in the testing procedure and more generally after the completion of each *put()* method to control whether the corresponding write operation succeeded. Any unexpected failure in the compare methods must be interpreted as a detected fault in the write functionality.

Figure 7 and Figure 8 show the SystemC TLM description for *t_peek()* and *t_compare()* respectively. *t_peek()* is a modification of *peek()* from the TLM library which reads the most recently written element of FIFO.

```
//t_peek()
template < typename T>
inline
T
testable_tlm_fifo<T>::t_peek(tlm_tag<T> *)
const
{
    //While (FIFO == Empty) Wait
    while (!tlm_fifo<T>::nb_can_get())
    {
        const_cast< testable_tlm_fifo<T> *>
        (this)->wait(m_data_written_event);
    }
    return
        buffer->peek_data(m_num_readable-1);
}
```

Figure 7 *t_peek()* SystemC TLM Description

```
//t_compare()
template < typename T>
inline
bool
testable_tlm_fifo<T>::t_compare
            (const T& val1_,const T& val2_){
    if (val1_ == val2_)
        return true;
    else {
        cout << "Fault Detected" << endl;
        return false;
    }
}
```

Figure 8 *t_compare()* SystemC TLM Description

After performing the first four operations, the FIFO is full, and we have to test its correct behavior in this condition. At this point, Step 5 of the Writer side tries to write an additional element in the FIFO. This operation should block the WRITER and the value (E) should not be written. The Compare method at Step 5 of the FIFO side should return false to check the correct blocking operation. Since this is an expected failure in the compare operation, it will not be considered as a fault.

At this point, in order to test that the writer is correctly unblocked when a value is read from the FIFO, we have to simulate a get operation. For this purpose, we introduce an additional test method:

- ***t_get():*** it works exactly as the *get()* method but internally to the FIFO.

Figure 9 shows the SystemC TLM description for *t_get()*. Steps 6 and 7 of the FIFO side read the top value from the buffer (it should be value A) by issuing a *t_get()* and then checking that the writer is correctly unblocked and enabled to write value E using a compare operation.

At this point, steps 8 to 11 of the FIFO side are used to empty the FIFO and to obtain again an empty channel by successively calling *t_get()* methods. When the FIFO gets

empty again, step 12 of FIFO side performs an additional *t_get()* to check the correctness of the blocking read operation. Since the *t_get()* method is blocking, the FIFO starts waiting for a new value. Step 6 of the WRITER side writes this value and operations 13 and 14 of the FIFO side check that this value is correctly stored in the channel. Finally the same procedure is repeated but using a testing version of the peek method (*t_peek()*) in steps 7 of the WRITER and 15 of the FIFO.

```
//t_get()
template < typename T>
inline
T
testable_tlm_fifo<T>::t_get( tlm_tag<T> * )
{
       return tlm_fifo<T>::get();
}
```

Figure 9 *t_get ()* SystemC TLM Description

The complete procedure for the WRITER and FIFO sides can finally be translated into additional SystemC methods in both sides for implementing TDG and TRE units. In the WRITER side, TDG is a method running in test mode which generates Test Data by issuing the *put()* transactions introduced in Figure 6. Figure 10 shows a possible SystemC TLM description for a writer with two Normal and Test modes. The WRITER side procedure can be re-used for any other writers by automatically adding the proposed test method to their descriptions.

```
void Writer::run()
{
    //Normal Mode
    if (N_T_mode == 0){
        //Normal Operation
    }
    //Test Mode
    else
    {
        //Steps 1-4: While FIFO is not Full
        while (write_port->nb_can_put())
        {
            in >> testData;
            write_port->put(testData);
        }
        //Step 5: Full -> Full(Blocked Writer)
        in >> testData;
        write_port->put(testData);

        //Step 6: Empty(Blocked Reader)->Empty
        in >> testData;
        write_port->put(testData);

        //Step 7: Empty(Blocked Reader)->SemFull
        in >> testData;
        write_port->put(testData);
    }
}
```

Figure 10 SystemC TLM Description of Writer

Figure 11 shows the SystemC TLM description of the *testable_tlm_fifo* class which inherits from the *tlm_fifo* primitive channel and implements the proposed testing facilities for testing the *put()* method as the TRE of FIFO. *testable_tlm_fifo* has an extra internal buffer (*m_test_buff*) to store test data used during *t_compare()* operations to check the writing functionalities. The previously introduced testing methods (*t_peek(), t_compare()* and *t_get()*) are defined in this class and are called by the overridden *put()* method which implements the FIFO side test responses.

```
template <class T>
class testable_tlm_fifo:public tlm_fifo<T>,
                        public sc_module
{
public:
    SC_HAS_PROCESS(testable_tlm_fifo);
    testable_tlm_fifo(sc_module_name name,
                      bool mode = false,
                      int size = 1):
    sc_module(name),
    tlm_fifo<T>(size)
    {
        m_N_Tmode = mode;
        m_test_size = size + 3;
        m_test_buf = new T[m_test_size];
        m_test_num = 0;
        SC_THREAD(test_unblock_writer);
        SC_THREAD(test_unblock_reader);
    }
    //Overridden put
    void put( const T& );

protected:
    void test_unblock_writer();
    void test_unblock_reader();

    T t_peek( tlm_tag<T> *t = 0 ) const;
    bool t_compare (const T&, const T&);
    T t_get( tlm_tag<T> *t = 0 );

    bool m_N_Tmode;
    int m_test_size;
    T*  m_test_buf;
    int m_test_num;
    sc_event fifo_full_event;
    sc_event fifo_empty_event;
};
```

Figure 11 *testable_tlm_fifo* Class Description

We also defined similar test procedures and methods for other *tlm_fifo* operations.

Table 1 shows the test procedures for the *get()* and the *peek()* functions of *tlm_fifo* as examples of Read Transaction Test Strategy. In these procedures only the Reader and the FIFO are involved in test. As for the test procedure of Figure 6 an additional test method is defined to simulate the *put()* method internal to the FIFO:

- ***t_put():*** works exactly as the *put()* method but it is internal to the FIFO.

Table 1 Test Procedures for *get()* and *peek()* Methods

get()		*peek()*	
READER	FIFO	READER	FIFO
x = get(A)	-	x = peek(A)	-
-	t_put(A)	-	t_put(A)
t_compare (x, A)	-	t_compare (x, A)	-
-	t_put(B)	-	t_put(B)
t_compare (get(),B)	-	t_compare (peek(),A)	-
-	t_put(C)	-	t_put(C)
-	t_put(D)	t_compare (peek(),A)	-
t_compare (get(),C)	-	-	t_put(D)
-	t_put(E)	t_compare (peek(),A)	-
-	t_put(F)	-	t_put(E)
t_compare (get(),D)	-	t_compare (peek(),A)	-
-	t_put(G)	-	t_get()
-	t_put(H)	t_compare (peek(),B)	-
-	t_put(I)	-	t_get()
t_compare (get(),E)	-	t_compare (peek(),C)	-
t_compare (get(),F)	-	-	t_get()
t_compare (get(),G)	-	t_compare (peek(),D)	-
t_compare (get(),H)	-	-	t_get()
t_compare (get(), I)	-	t_compare (peek(),E)	-
		-	t_get()

7. Test Quality Evaluation

To evaluate the proposed testing strategies, early quality evaluation metrics are needed during the design phase. These metrics should be easily measurable, available and acceptable at this very high level of abstraction and should be re-useable on the final product with the same semantic and accuracy.

Performing test coverage/quality evaluation at very high abstraction levels is always a complex task. At this level, information about the actual implementation of the elements composing the system are completely missing (the same component may lead to a software routine or to a hardware element). For this reason any tentative of using coverage metrics defined for different abstraction levels will probably lead to erroneous results. In order to evaluate the quality of the proposed test strategy we have to introduce coverage metrics at the same abstraction level used to model our *tlm_fifo*.

Concerning high-level techniques, at a first look, similarities can be found with functional verification approaches and their related coverage metrics. Actually, the basic difference with respect to functional verification is that, since we are dealing with testing, our goal is to "embed" the testing procedures directly into the system itself. On the other hand, functional verification is usually performed using an external verification environment providing a much higher set of verification capabilities. Evaluating the effectiveness of our approach in terms of coverage metrics used to evaluate verification sequences generated by functional verification tools such as SpecMan™ from Cadence is thus not a viable solution. The results would be probably affected by the different evaluation conditions.

The only feasible solution to perform our test quality analysis is thus to identify new metrics. In order to avoid the definition of new custom metrics, being TLM descriptions basically C++ programs, we decided to adopt metrics widely used in the software community.

The coverage metric we exploit in this paper is related to the state chart model of the *tlm_fifo* introduced in Section 5. The proposed test procedure was developed by trying to stress each method of the FIFO in different operational conditions. Since calling a method is equivalent to firing a transition on the state chart model of the FIFO, a possible measure concerns checking whether the test is able to fire all the possible transitions and to reach all possible states. This measure can be used as a coverage metric.

Table 3 reports the coverage evaluation for the test procedure of the *put()* method proposed in Figure 6. For each step of the testing procedure we report the actual content of the FIFO, the initial state, the fired transition and finally the final state. The initial state and the final state are identified with the corresponding state numbers shown in Figure 5, whereas the transition is identified by the corresponding method call placed on the arc. A consideration is required regarding transitions. The introduction of the testing methods introduces additional transitions. These transitions are perfectly equivalent to the one already reported in Figure 5 for the equivalent normal methods (i.e., an additional transition for the *t_get()* method is introduced for every transition labeled with the *get()* method).

Looking at the results of Table 3 we can conclude that the proposed procedure allows reaching 100% of coverage of the FIFO states (it reaches each state of the state diagram of Figure 5). Concerning the coverage of the transitions we have 28 normal mode transitions in the diagram of Figure 5. The test of the *put()* method covers 6 of these transitions (the remaining used transitions are testing mode transitions) allowing to reach 21.4% coverage of all transitions. Obviously the remaining transitions have to be covered by the test procedures of the remaining methods.

We performed the same evaluation for the test procedures defined for the remaining *tlm_fifo* methods. Table 2 summarizes the results of this evaluation. For each method we provide state coverage, transition coverage, and finally the number of state transitions required to perform the test. The latter can be considered as a cost factor for the given test procedure. By looking at the full set of test procedures

we have been able to reach 100% of both states and transition coverage.

Table 2 Coverage Evaluation

tlm_fifo Methods	State Coverage	Transition Coverage	Number of State Transitions
put()	100%	21.4%	23
nb_put()	83.3%	21.4%	24
nb_can_put()	83.3%	17%	24
get()	83.3%	17%	19
nb_get()	66.6%	17%	25
nb_can_get()	66.6%	14.2%	21
peek()	83.3%	14.2%	20
nb_peek()	66.6%	14.2%	29
nb_can_peek()	66.6%	14.2%	21

7. Conclusions

This paper presented the first steps towards the definition of a design methodology capable of guaranteeing the implementation of "Plug & Test" (test-enriched) communication channels at the TLM level. The proposed methodology relies on introducing additional test functionalities to the blocks composing a TLM design to be translated later on into *Built-in Functional Self Test* (BIFST) facilities in the final product.

The paper focused on the definition of a testable *tlm_fifo*, representing one of the main TLM primitives. We gave an overview of the approach we followed to add test functionalities to *tlm_fifo,* trying to highlight how the same methodology can be applied to other TLM primitives with the goal of defining a complete system level library of "Plug & Test" primitives.

To evaluate the quality of the proposed test strategies, we presented the results obtained by applying coverage metrics widely used in the software community to the proposed problem. These high-level evaluation metrics also have the capability of being re-used in the final product with the same semantics and accuracy.

8. Acknowledgements

We acknowledge the contributions of Marco Cimei from Politecnico di Torino who significantly helped us in this work.

9. References

[1] T. Grötker, S. Liao, G. Martin, S. Swan, System Design with SystemC. Springer, 2002, Chapter 8, pp. 131.

[2] Frank Ghenassia (editor), Transaction-level Modeling with SystemC: TLM Concepts and Applications for Embedded Systems, Springer, 2005.

[3] M. Abramovici, M. Breuer, and A. Friedman. Digital SystemsTesting and Testable Design. IEEE Press, 1990.

[4] L. Lingappan and N. K. Jha. Unsatisfiability based Efficient Design for Testability Solution for Register-Transfer Level Circuits. In Proc. VLSI Test Symposium, pages 418–423, 2005.

[5] D. Xiang and J. H. Patel, "Partial scan design based on circuit state information," IEEE Trans. Computers, vol. 53, pp. 276–287, Mar. 2004.

[6] X. Lin, I. Pomeranz, and S. M. Reddy, "Full scan fault coverage with partial scan," in Proc. Design, Automation & Test Europe Conf., pp. 468–472, Mar. 1999.

[7] K. T. Cheng and V. D. Agarwal, "A partial scan method for sequential circuits with feedback," IEEE Trans. Computers, vol. 39, pp. 544–548, Nov. 1990.

[8] S. T. Chakradhar, A. Balakrishnan, and V. D. Agrawal, "An exact algorithm for selecting partial scan flip-flops," in Proc. Design Automation Conf., pp. 81–86, June 1991.

[9] V. Chickermane and J. H. Patel, "A fault oriented partial scan design approach," in Proc. Int. Conf. Computer-Aided Design, pp. 400–403, Nov. 1991.

[10] M. S. Hsiao, G. S. Saund, E. M. Rudnick, and J. H. Patel, "Partial scan selection based on dynamic reachability and observability information," in Proc. Int. Conf. VLSI Design, pp. 174–180, Jan. 1997.

[11] V. Boppana and W. K. Fuchs, "Partial scan based on state transition modeling," in Proc. Int. Test Conf., pp. 538–547, Oct. 1996.

[12] V. S. Iyengar and D. Brand, "Synthesis of pseudo-random pattern testable designs," in Proc. Int. Test Conf., pp. 501–508, Aug. 1989.

[13] B. H. Seiss, P. M. Trouborst, and M. H. Schulz, "Test point insertion for scan-based BIST," in Proc. European Test Conf., pp. 253–262, Apr. 1991.

[14] N. A. Touba and E. J. McCluskey, "Test point insertion based on path tracing," in Proc. VLSI Test Symp., pp. 2–8, Apr. 1996.

[15] I. Ghosh, A. Raghunathan, and N. K. Jha, "A design for testability technique for register-transfer level circuits using control/data flow extraction," IEEE Trans. Computer-Aided Design, vol. 17, pp. 706–723, Aug. 1998.

[16] S. Ravi, G. Lakshminarayana, and N. K. Jha, "TAO: Regular expression-based register-transfer level testability analysis and optimization," IEEE Trans. VLSI Systems, vol. 9, pp. 824–832, Dec. 2001.

[17] B. T. Murray and J. P. Hayes: "Hierarchical test generation using pre computed tests for modules," IEEE Trans. on CAD, Vol. 9, No. 6, pp. 594–603, June 1990

[18] H. Wada, T. Masuzawa, K. K. Saluja and H. Fujiwara, "Design for strong testability of RTL data paths to provide complete fault efficiency," Proc. 13th Int. Conf. on VLSI Design, pp. 300–305, 2000.

[19] S. Ohtake, H. Wada, T. Masuzawa and H. Fujiwara: "A non-scan DFT method at register-transfer level to achieve complete fault efficiency," in Proc. of Asian

South Pacific Design Automation Conference (ASP-DAC), pp. 599–604, 2000.
[20] I. Ghosh, A. Raghunath and N. K. Jha: "Design for hierarchical testability of RTL circuits obtained by behavioral synthesis," in Proc. of IEEE Int. Conf. on Computer Design, pp. 173–179, 1995.
[21] S. Boubezari and et. al. Testability Analysis and Test-Point Insertion in RTL VHDL Specifications for Scan-Based BIST. IEEE Transactions on CAD, 18(9):1327– 1340, Sept. 1999.
[22] J. Carletta and C. Papachristou. Testability Analysis and Insertion for RTL Circuits Based on Pseudorandom BIST. In Proc. IEEE International Conference on Computer Design, pages 162–167, 1995.
[23] S. Roy, G. Guner, and K.-T. Cheng. Efficient Test Mode Selection and Insertion for RTL-BIST. In Proc. of International Test Conference, pages 263–272, 2000.
[24] L. Fang, NEC Laboratories America and K.J. Balakrishnan, NEC Laboratories America, "RTL Test Point Insertion to Reduce Delay Test Volume", Proceedings of VLSI Test Symposium 2007
[25] An approach for redesign for testability at the register-transfer level Harmanani, H M; Harfoush, S, CAN J ELECTR COMPUT ENG. Vol. 25, no. 4, pp. 163-168. Oct. 2000.
[26] Zhiqiang You, Ken-ichi Yamaguchi, Michiko Inoue, Jacob Savir, Hideo Fujiwara: Power-Constrained Test Synthesis and Scheduling Algorithms for Non-Scan BIST-able RTL Data Paths. IEICE Transactions 88-D(8): 1940-1947 (2005).
[27] OSCI SystemC TLM 2.0 Standard, http://www.systemc.org/projects/tlm/document/TLM_2.0_Overview/en/1
[28] S. Mirkhani, Z.Navabi, System Level Design Languages, The VLSI Handbook, Chapter 86, CRC Press, 2nd Edition, Dec. 2006.
[29] A. Rose, S. Swan, J. Pierce, J.-M. Fernandez, Transaction Level Modelling in SystemC, OSCI white-paper, 2004.

Table 3 Coverage Evaluation for the *put()* Method

WRITER	FIFO	FIFO Content	Initial State	Transition	Final State
put(A)	-	A	0	put	3
-	t_compare (t_peek(), A)	A	3	t_peek	3
put(B)	-	B A	3	put	3
-	t_compare (t_peek(), B)	B A	3	t_peek	3
put(C)	-	C B A	3	put	3
-	t_compare (t_peek(), C)	C B A	3	t_peek	3
put(D)	-	D C B A	3	put	4
-	t_compare (t_peek(), D)	D C B A	4	t_peek	4
put(E)	-	D C B A	4	put	5
-	t_compare (t_peek(), E)	D C B A	5	t_peek	5
-	t_compare (t_get(), A)	E D C B	5	t_get	4
-	t_compare (t_peek(), E)	E D C B	4	t_peek	4
-	t_compare (t_get(), B)	E D C	4	t_get	3
-	t_compare (t_get(), C)	E D	3	t_get	3
-	t_compare (t_get(), D)	E	3	t_get	3
-	t_compare (t_get(), E)		3	t_get	0
-	x = t_get(F)		0	t_get	2
put(F)	-		2	put	0
-	t_compare (x, F)		0	-	0
-	x = t_peek(G)		0	t_peek	1
put(G)	-	G	1	put	3
-	t_compare (x, G)	G	3	-	3
-	t_compare (t_get(), G)		3	t_get	0